Data Pump Architecture Simulator and Performance Model

A Thesis

Submitted to the Faculty

of

Drexel University

by

Douglas F. Jones, Jr.

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Science

June 2010

Acknowledgements

I'd like to express my most sincere gratitude to my advisor, Dr. Jeremy Johnson, for his mentoring and advice for the production of this thesis and throughout the entirety of my studies and research while at Drexel. I'd like to thank everyone involved with the DPA project, especially Professors Markus Püschel, Franz Franchetti, and James C. Hoe. A special thanks is reserved for Qian Yu, whose insight and aid throughout my work on the DPA project and this thesis was invaluable.

I am grateful for the friendship and advice of my colleagues in our lab, including Tim Chagnon, Gavin Harrison, Danh Nguyen, and Lingchuan Meng. Finally, I'd like to thank my friends and family, especially my mother, father, and brother, for their love and support throughout my college career.

Table of Contents

List of Tables

List of Figures

Abstract

Data Pump Architecture Simulator and Performance Model

Douglas F. Jones, Jr.

Advisor: Jeremy Johnson, PhD

The Data Pump Architecture (DPA) is a novel non-von-Neumann computer architecture emphasizing efficient use of on-chip SRAM and off-chip DRAM bandwidth. The DPA is parameterized by local memory size, memory bandwidth, vector length, and number of compute processors, allowing the architecture configuration to be balanced for a given set of computations. The Data Pump Architecture Simulator and Performance Model is a functional simulator providing an implementation of the DPA as a software library. Simulation at this level provides the user with the ability to test algorithms for correctness as well as performance. A hardware designer may then use the Simulator as a tool to experimentally determine the effects of architecture parameters and software algorithms through performance data provided by the Simulator's model. The benefit of the approach described here compared to alternatives such as logic/gate level or RTL/HDL software emulation is primarily a reduction in the time needed for 1) a designer to modify and compile a HDL design and 2) to perform the simulation. The Simulator serves as a bridge between the application specific hardware and software design. The result is the ability to use the simulator as part of a framework for rapidly investigating the construction of an optimal system architecture for a given set of algorithms. Investigations with the Walsh-Hadamard transform (WHT), a prototypical digital signal processing transform, are shown. The SPIRAL (www.spiral.net) code generation system is used to explore the space of WHT algorithms while the Simulator is used to explore the performance and trade-offs of different DPA configurations.

1. Introduction

1.1 Motivations

Simulation is an important part of architecture design and analysis. Many modern software tools exist allowing hardware to be simulated with high accuracy in terms of functional replication as well as performance prediction. These tools typically accept input in the form of a RTL/HDL description such as Verilog or VHDL. However, simulation at this level of detail is expensive in terms of development time for HDL descriptions and execution time to simulate the model. Functional and other types of high-level simulators require less execution time for model simulation but with reduced accuracy for performance estimates and, perhaps, functional correctness. For rapid architecture exploration, the ideal solution would produce accurate performance estimates with short execution times.

Architecture exploration is of interest because there is a fundamental limit on the resources allocated amongst the components of a computer system. There are likely to be constraints on the power and/or physical area consumed by a system. Under such constraints the question of where to allocate these resources becomes prudent. To physically produce hardware for every combination of architecture parameters of interest is infeasible. Therefore, simulation is necessary for the designer to choose parameters that will create a balanced and efficient system.

The Data Pump Architecture serves as a platform for efficient embedded digital signal processing applications. As such, it is parameterized so that configurations can be selected to create a balanced architecture for the target application. Traditional RTL simulators provide architecture simulation. However, they offer little in terms of feedback for software development and optimization, leaving a gap between the

ISA and the interface the software developer will target. In the context of the DPA, the SPIRAL code generation system serves as the primary "developer" for software targeted to the DPA. Given such a code generation system, a tool to serve as the bridge between the software and hardware is desirable. Such a tool can then be used as an aide to algorithm verification and as a source of feedback for effects of algorithm optimization.

1.2 Related Work

There are numerous high level simulator systems in common use today. Virtutech Simics is a full system simulator capable of simulating heterogeneous architectures [20]. Flexus, part of the SimFlex project, is a flexible component-based simulator built on top of Simics [11]. Flexus provides the designer with the capability to select modules that implement different system components and compose them to build a complete simulation system. PTLSim is a cycle accurate simulator for the x86 family of processors [9]. In combination with virtualization technology, PTLsim is able to simulate multi-thread and multi-processor systems with high performance. Asim is a modular, performance model framework [3]. Modules map to architectural components in the architecture under study. ManySim is a modular simulation system designed for exploring large-scale chip multiprocessor architectures [21]. ManySim is trace driven and supports modules that represent cores, interconnects, caches, and memory systems.

1.3 Result Summary

This thesis presents a high-level, functional simulator and performance model for the Data Pump Architecture. The Simulator has been verified using common Digital Signal Processing (DSP) algorithms, such as implementations of the Walsh-Hadamard

and fast Fourier transforms. These algorithms have also been used to benchmark the Simulator's performance model through comparison to HDL simulation using ModelSim. Finally, the DPA Simulator is used to perform an exploration of the architecture parameters available on the DPA, showing the effect of parameter choice on the performance of an example DSP algorithm, the Walsh-Hadamard transform.

1.4 Organization of the Thesis

This thesis is organized as follows. Chapter 2 provides background on algorithms of interest and their mathematical notation in addition to a discussion of commodity architectures and the SPIRAL code generator. Chapter 3 describes the Data Pump Architecture at the functional unit and ISA level including a section on programmer considerations. Chapter 4 describes the DPA Simulator's design, implementation, and how it enables generation of DPA assembly from C code. Chapter 5 describes the DPA Simulator's Performance model including accuracy information and benchmarks. Chapter 6 shows an example architecture exploration using the Walsh-Hadamard transform as the target application. Chapter 7 contains the conclusion.

2. Background

2.1 Linear Transformations for Digital Signal Processing

In the domain of Digital Signal Processing, most calculations of interest can be expressed by fast discrete linear transformation of the form

$$y = A\ x \tag{2.1}$$

where A is the transformation matrix, x is the input vector, and y is the output vector. Transformations of this form have important computational properties. Data access patterns for fixed input size are constant, meaning the algorithm is data independent. The transformation matrix A can be factored into highly structured sparse matrices, allowing smaller blocks of computation to be used to build the entire transformation in a divide-and-conquer fashion [12].

High performance code can be created by taking advantage of these inherent properties. To do so, the Kronecker product is used as a mathematical formalism for the representation of matrix factorizations. The transformations can then be manipulated analytically to exhibit beneficial computational properties with respect to the properties of the targeted computer architecture. From the analytical representation, high performance code can be generated automatically.

2.1.1 Kronecker/Tensor Product

For certain linear transforms, it is convenient to use the Kronecker (or tensor) product as a tool to factor a larger matrix into two smaller matrices. In addition, the properties of this operation allow matrix equations to be manipulated into forms with desirable mathematical properties that correspond to different implementation

choices.

The Kronecker product, $A \otimes B$, of two matrices A and B of sizes $m \times n$ and $p \times q$ respectively is defined such that the resulting matrix is defined by

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix}. \tag{2.2}$$

The size of the resulting matrix is $mp \times nq$. Intuitively, the resulting matrix is built by multiplying each element of A to the entire matrix B.

The Kronecker product has several properties that can be used to manipulate Kronecker equations into convenient forms [6]. We have the following results when taking the product with identity matrices

$$y = (I_p \otimes A)\ x = \begin{bmatrix} A & & \\ & \dots & \\ & & A \end{bmatrix} x \tag{2.3}$$

$$y = (A \otimes I_q)\ x = \begin{bmatrix} a_{11}I_q & \dots & a_{1n}I_q \\ \vdots & \dots & \vdots \\ a_{m1}I_q & \dots & a_{mn}I_q \end{bmatrix} x. \tag{2.4}$$

Equation 2.3 can be thought of as a loop over a parallel operation where A is applied to p independent segments of the input vector x. Equation 2.4 can be thought of as a vector operation where A is applied to vectors of length q. More precisely, let X_{jq} denote the subvector of x of length q starting at index jq and similarly for Y_{iq}. Then

the computation given by Equation 2.4 is

$$Y_{iq} = \sum_{j=1}^{n} a_{ij} X_{jq}, \quad i = 1, \ldots, m \tag{2.5}$$

The multiplication $a_{ij}X_{jq}$ is a scalar-vector product and the sum involves vector addition of length q.

The Kronecker product is associative

$$(A \otimes B) \otimes C = A \otimes (B \otimes C), \tag{2.6}$$

but not commutative thus

$$A \otimes B \neq B \otimes A. \tag{2.7}$$

It is distributive over addition,

$$(A + B) \otimes (C + D) = (A \otimes C) + (A \otimes D) + (B \otimes C) + (B \otimes D). \tag{2.8}$$

The multiplicative property

$$(AC) \otimes (BD) = (A \otimes B)(C \otimes D) \tag{2.9}$$

can be used to derive many useful expressions including

$$A \otimes B = (A \otimes I)(I \otimes B) \tag{2.10}$$

$$= (I \otimes B)(A \otimes I). \tag{2.11}$$

In addition, it is important to note the special case

$$I_m \otimes I_n = I_{mn}. \tag{2.12}$$

2.1.2 Walsh-Hadamard Transform

The Walsh-Hadamard Transform can be computed by the matrix-vector product

$$W_N \cdot x \tag{2.13}$$

where $N = 2^n$ and x is the input vector.

The matrix W_N can be defined recursively

$$\begin{aligned} W_N &= W_{N-1} \otimes W_2 && (2.14)\\ &= \underbrace{W_2 \otimes \ldots \otimes W_2}_{n} && (2.15) \end{aligned}$$

where

$$W_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.16}$$

and $\otimes$ is the Kronecker product.

For example, the matrix W_4 can be written as

$$\begin{aligned} W_4 &= W_2 \otimes W_2 && (2.17)\\ &= \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} && (2.18)\\ &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}. && (2.19) \end{aligned}$$

2.1.3 WHT Algorithms

The properties of the Kronecker product and the definition of the WHT can be used to create multiple factorizations of the WHT matrix [7]. These factorizations can be created such that they allow for particular implementations of the transform.

Recursive Factorization

A WHT matrix of size $N = 2^n$ can be factored such that the factors contain successively smaller WHT matrices. In the following equations, the subscripts, k as in A_k, will be powers of two such that W_N is the WHT matrix of size 2^n and $W_{N-1} = W_{2^{n-1}}$.

Then, we can write

$$\begin{aligned} W_N &= W_{N-1} \otimes W_2 && (2.20) \\ &= W_{N-1} I_{N-1} \otimes I_2 W_2 && (2.21) \\ &= (W_{N-1} \otimes I_2)(I_{N-1} \otimes W_2) && (2.22) \end{aligned}$$

using Equations 2.9 and 2.14. This factorization leads to a recursive implementation where a WHT of size N is defined by two WHTs of size $N - 1$ and 2.

Iterative Factorization

A different factorization leads to an iterative implementation. For $W_N = W_{2^n}$ we have

$$
\begin{aligned}
W_N &= \bigotimes_{i=1}^{n} W_2 && (2.23)\\
&= (W_2 \otimes I_{2^{n-1}})(I_2 \otimes \bigotimes^{n-1} W_2) && (2.24)\\
&= (W_2 \otimes I_{2^{n-1}})(I_2 \otimes (W_2 \otimes I_{2^{n-2}})(I_2 \otimes \bigotimes^{n-2} W_2)) && (2.25)\\
&= (W_2 \otimes I_{2^{n-1}})(I_2 \otimes W_2 \otimes I_{2^{n-2}})(I_4 \otimes \bigotimes^{n-2} W_2) && (2.26)\\
&= (W_2 \otimes I_{2^{n-1}})(I_2 \otimes W_2 \otimes I_{2^{n-2}})(I_4 \otimes (W_2 \otimes I_{2^{n-3}})(I_2 \otimes \bigotimes^{n-3} W_2) && (2.27)\\
&= (W_2 \otimes I_{2^{n-1}})(I_2 \otimes W_2 \otimes I_{2^{n-2}})(I_4 \otimes W_2 \otimes I_{2^{n-3}})(I_8 \otimes \bigotimes^{n-3} W_2) && (2.28)\\
&= \prod_{i=1}^{n} (I_{2^{i-1}} \otimes W_2 \otimes I_{2^{n-i}}) && (2.29)
\end{aligned}
$$

using Equation 2.9 to perform the expansion. This leads to $n = \log_2(N)$ factors containing the W_2 matrix and is labeled as an iterative algorithm since the factors can be applied iteratively to the data vector.

Parallel Factorizations

In addition to recursive and iterative factorizations, Kronecker equations can be thought of as providing parallel and vectorized implementations [6]. Equations that allow for parallel implementations have the form

$$y = (I_m \otimes A)x \tag{2.30}$$

which naturally corresponds to a loop of m iterations where A is applied to the input vector. For example, if the input is x and the output is y then we would have the pseudo-code

$$\text{for} \quad i = 0 \ldots m-1$$
$$A(y(ni), x(ni))$$

where the transform A with size n is applied to vectors accessed at index ni. Each iteration of the loop can be executed in parallel since the input and output indices do not overlap.

The granularity of the parallel execution can be controlled by manipulating the size of the identity matrix. For instance, if we have $I_{mn} \otimes A$ we can rewrite this as $I_m \otimes (I_n \otimes A)$ which would allow m processors can execute $I_n \otimes A$ in parallel.

Vector Factorizations

Equations that allow for vector implementations have the form

$$y = (A \otimes I_v)x \tag{2.31}$$

which naturally corresponds to an implementation using vectors of length v. As an example, if we have $y = (A \otimes I_2)x$ and we take A to be W_2 then we have the following

$$\begin{aligned}
\mathrm{y}(0) &= \mathrm{x}(0) + \mathrm{x}(2) \\
\mathrm{y}(1) &= \mathrm{x}(1) + \mathrm{x}(3) \\
\mathrm{y}(2) &= \mathrm{x}(0) - \mathrm{x}(2) \\
\mathrm{y}(3) &= \mathrm{x}(1) - \mathrm{x}(3)
\end{aligned}$$

which is equivalent to two applications of W_2 at stride 2. If the operations are grouped and we assume the availability of vectors of two elements, then we have

```
v1 = load(x(0))
v2 = load(x(2))
t1 = v1 + v2
t2 = v1 - v2
store(t1, y(0))
store(t2, y(2))
```

where load and store are used to read and write vectors of length 2.

Thus, the Kronecker product properties can be used to factor large transforms in a way that matches the parameters (execution units, vector registers) of a given machine. For example, we can favor equations with terms of the form $(I_m \otimes A \otimes I_v)$ for machines with m parallel units and vectors of v elements.

General Factorization

The iterative and recursive factorization can be generalized. The generalized factorization, for n as powers of 2 and $n = n_1 + \cdots + n_t$, is

$$W_N = \prod_{i=1}^{t} (I_{n_1+\cdots+n_{i-1}} \otimes W_{n_i} \otimes I_{n_{i+1}+\cdots+n_t}). \tag{2.32}$$

This representation can be used to create any of the previously mentioned factorizations. Equation 2.32 mixes various amounts of recursion and iteration and allows the exploration of a large space of WHT algorithms. Furthermore, 2.32 can be manipulated to account for implementation considerations. Considering on-chip memory size, we can select W_{n_i} to be sized such that the input for this stage of the algorithm can fit into half of on-chip memory. While computation is performed on the first half of memory, data can be stored and loaded to the second half, resulting in a factorization suited for software pipelining techniques. Using this approach, the access stride

must be managed such that it does not grow too large given the capacity of on-chip memory, thus reducing the total number of transfers needed per input element.

2.2 Commodity Architectures

Commodity CPUs, such as those implementing the x86 and x86_64 ISAs, often feature superscalar, pipelined architectures with large caches shared by multiple cores on a single chip. These designs evolved out of the need to provide good performance across a wide variety of common applications. However, for specialized applications, such as the linear transforms discussed here, commodity architectures are inefficient. Some algorithms, such as the discrete Fourier transform (DFT) can reach only 50% of the peak performance on modern CPUs [19].

With commodity CPUs, much attention must be paid to the memory hierarchy, especially the on-chip cache. This is due to the increasing gap between the speed of processor execution and transfers from main system memory to CPU registers [19]. In these architectures, the cache operates by assuming spatial and temporal locality, allowing it to implicitly determine what portions of the address space will be stored on chip. Modern caches use a multi-way set associative design with as many as three levels of increasing capacity. Knowing this, code can be optimized with respect to cache utilization by increasing reuse (the number of calculations on a single piece of data) and spatial locality (accessing data nearby in the address space).

Cache design parameters are selected to minimize misses and maximize utilization for general use cases and may not provide high utilization for all algorithms. The result is that there is a limit to the extent of optimization possible given the algorithm. For instance, matrix multiplication has a reuse degree of $O(n)$ while the DFT only has a degree of $O(\log(n))$ [19]. For algorithms with low degrees of reuse, high levels of performance relative to the peak performance can only be achieved for problem

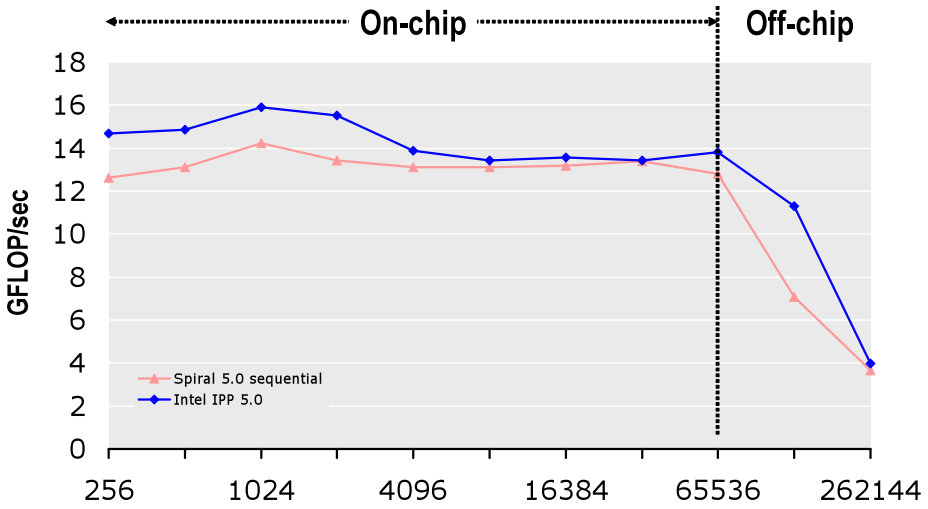

Figure 2.1: Single Precision DFT Performance, 4-way vectors, 3GHz Intel Core2 [14]

sizes smaller than the amount of cache. As the problem size increases and data spills from the lowest level of cache to the upper levels and, eventually, main memory, performance decreases. As this occurs, the performance bottleneck begins to shift from integer/floating point execution capacity to bandwidth between memory and the CPU. Figure 2.1 shows single precision DFT performance on a commonly available CPU as the problem size increases. Here, high performance code from the Spiral generator and Intel IPP library [5] is shown. Both show a sharp drop in performance as the problem size grows larger than the available on-chip memory.

2.3 SPIRAL Code Generation

SPIRAL is a software system for code generation of high performance linear digital signal processing algorithms [8]. SPIRAL automatically tunes the algorithm's implementation for a given target architecture. This process is approached as an optimization problem such that the space of possible algorithm implementations is represented using a domain-specific mathematical structure similar to the Kronecker product notation discussed in Section 2.1.1. Using this notation, SPIRAL can explore the space of possible factorizations. Internally, SPIRAL represents the space as a tree structure, where the root is the original, full sized transform matrix and leaves represent the transform matrices that will be used in the final algorithm. SPIRAL optimizes algorithms using a feedback driven search engine over the algorithm space. As feedback is gained for each possible implementation, SPIRAL is able to inform the search engine of how to proceed.

SPIRAL accepts feedback in the form of algorithm execution time. When tuning for the host platform, the system's native timer utilities and/or processor level timer registers can be used to produce highly accurate timings. However, when the host system is not available, as might be the case for systems under development, a system simulator or performance model can be used to estimate the execution time and provide feedback to the SPIRAL system.

SPIRAL is able to take advantage of an architecture's features when generating code by making use of multiple processing elements and/or cores as well as vector instructions. On modern architectures, instruction scheduling, the use of vector instructions, and memory access pattern optimization is critical for achieving performance approaching the peak performance for a given architecture [19]. SPIRAL approaches such systems by providing the capability to tag algorithms with details of the system's architecture. This information might include, for instance, the number of

compute processors and the vector width. Given such information, SPIRAL will limit the search space to only those algorithms that are able to take advantage of the features described by the tags. In relation to the Kronecker Product notation discussed earlier, when using tags SPIRAL will only consider factorizations that can support the selected tags. When specifying processor number and vector width, SPIRAL will only consider factorizations that can be fully parallelized and vectorized.

SPIRAL can be used as part of a system for generating high performance code for the Data Pump Architecture. By enabling the SPIRAL system to search the DPA parameter space in conjunction with algorithm optimization, a global optimal hardware/software system can be created with respect to external constraints.

3. Data Pump Architecture

3.1 Architecture Overview

The Data Pump Architecture (DPA) is a non-von Neumann architecture for embedded digital signal processing applications [18]. The DPA is part of the SPIRAL project [14]. Members of the DPA project include those from Carnegie Mellon University, University of Illinois at Urbana-Champaign, and Drexel University. The DPA is designed to provide fast, efficient application specific computation. The DPA can be configured such that its microarchitecture is well suited to the intended application. The programmer is given full control over the memory subsystems, avoiding the cache utilization problems suffered on commodity architectures. In addition, the architecture is highly parameterized, allowing the architectures instantiation to be selected to match the memory and computational requirements of the targeted application. The DPA Instruction Set Architecture (ISA) is an extension of the SPARC V8 ISA [4] and is defined in the DPA ISA document and its addenda [18] [15] [16] [17]. The DPA implementation extends the open source LEON 3 by Gaisler [2].

The DPA's architecture diagram is shown in Figure 3.1. There are three computational units shown in the DPA architecture: the Data Processor (DP), the Compute

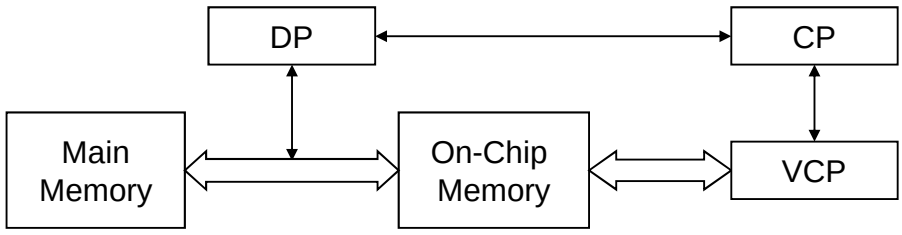

Figure 3.1: The DPA architecture [18].

Processor (CP), and the Vector Co-Processor (VCP). The DPA's system memory is divided into two layers: Main Memory and On-Chip Memory (referred to as Local Memory). Main Memory is located off chip. Transfers between Main Memory and Local Memory are directed by the Data Processor (DP). Transfers between Local Memory and the VCP's register file are directed by the CP. In addition, the CP issues instructions to the VCP.

3.2 SPARC Extension Overview

The primary processors in the DPA, the DP and CP, are implemented as extensions to the SPARC V8 ISA. Each processor of this type has its own private memory for instruction code and stack and a register file as described in the SPARC ISA [4]. DPA instructions are inlined with the SPARC instruction stream. When DPA instructions are decoded, they are handed off to the DPA's execution units [18]. The DPA extended SPARC processors are single issue. Every cycle one SPARC or one DPA instruction may be issued.

3.3 Memory

Memory in the DPA is divided into two levels: Main Memory (MEM) and Local Memory (LM).

3.3.1 Main Memory

Main Memory is located off chip and is generally classified as having greater capacity and access times than Local Memory. In implementations at the time of writing, Main Memory is DDR memory. Transfer operations from Main Memory move contiguous blocks of data to Local Memory.

3.3.2 Local Memory

Local Memory is characterized by having smaller capacity and access time than Main Memory. In addition, this memory is located on-chip. Local Memory is subdivided into segments, which may have their own type. Local Memory addresses are defined to be 32 bits long where the upper 5 bits are the segment number and the lower 27 bits are the segment offset. Access to Local Memory is always vector aligned. Local Memory contents can not be accessed directly by the Compute Processor. Instead, the CP is responsible for transferring data in LM to the vector register file where the data can then be processed. All transfers to and from Local Memory, including transfers to the register file, are non-blocking, meaning subsequent instructions will be issued before the transfer completes. Transfers are initially queued and scheduled for execution. When the transfer completes, the associated status register is updated. More detail on the transfer status registers can be found in Section 3.6.1 while detail on the transfer instructions is found in Section 3.7.

3.4 Data Processor

The Data Processor (DP) is an extended SPARC V8 core and is responsible for issuing data movement commands for transfers from Main Memory to Local Memory. The architecture of the DP can be seen in Figure 3.2. Here the DP is shown with its own private memory, as part of the SPARC implementation as well as its connections to Main Memory and Local Memory. The bi-directional mailbox transmit (TX) and receive (RX) connections are shown as well.

3.5 Compute Processor

The Compute Processor is an extended SPARC V8 core and is responsible for data transfers between Local Memory and the Vector Register File and issuing oper-

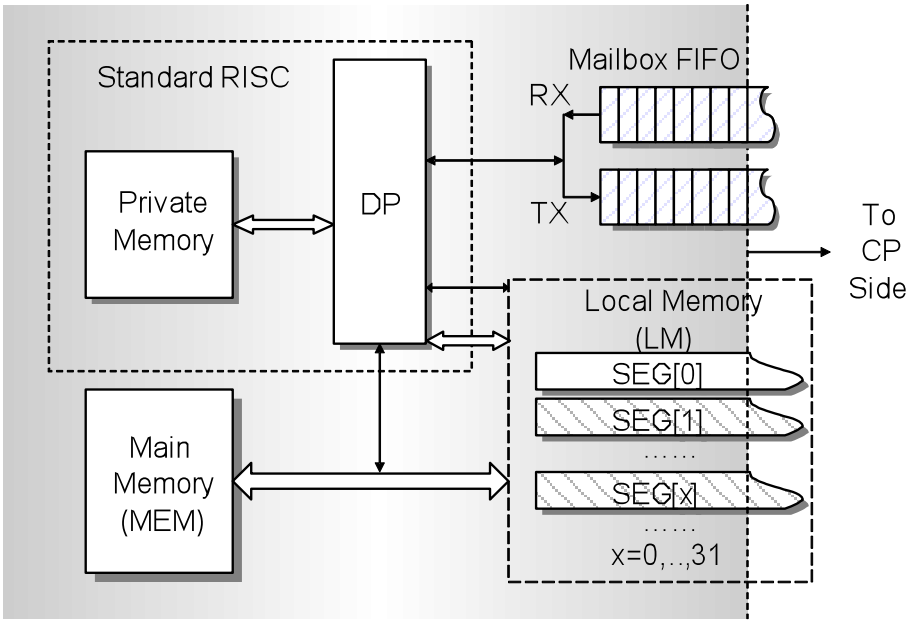

Figure 3.2: The DP Processor architecture [18]

ations to the Vector Co-Processor. The CP guides computation flow through these responsibilities. The architecture of the CP can be seen in Figure 3.3. Here the CP is shown with its own private memory, as part of the SPARC implementation as well as its connections to Local Memory and the vector register file (labeled VPR in the diagram). The floating point execution units, labeled FP, are attached to the vector register file. The CP's mailbox connection with transmit (TX) and receive (RX) channels are shown.

3.6 Processor Synchronization

There are two methods of inter-processor synchronization available in the DPA ISA. The first is through inspection of transfer counter registers. The second is through explicit communication using bi-directional mailboxes. In addition, intra-processor synchronization is provided in the form of a barrier instruction.

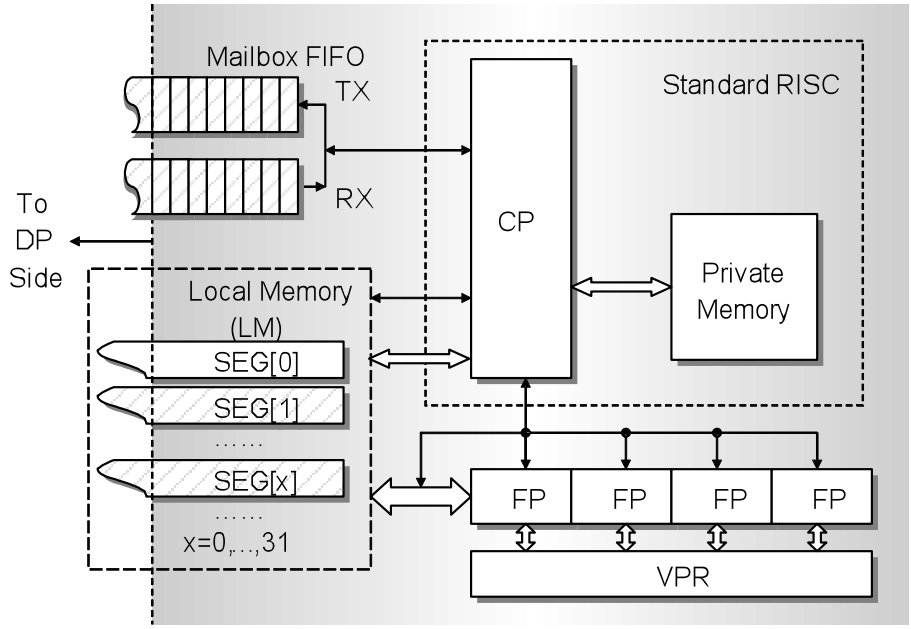

Figure 3.3: The CP Processor architecture [18]

3.6.1 Transfer Counters

Each Local Memory segment has a set of transfer counters associated with it. The counters and their purposes are as follows:

DWCNT DP-side Write Counter. This counter records the number of transfers from Main Memory to Local Memory.

DRCNT DP-side Read Counter. This counter records the number of transfers from Local Memory to Main Memory.

CWCNT CP-side Write Counter. This counter records the number of transfers from the Vector Register File to Local Memory.

CRCNT CP-side Read Counter. This counter records the number of transfers from Local Memory to the Vector Register File.

All counters are updated atomically as transfer instructions are retired. Basic synchronization may be performed by waiting for counters to obtain a certain value. This type of synchronization can be useful for data-independent algorithms that will execute a fixed number of transfers.

In addition to segment counters, the DPA system supports intra-processor mailbox communication. The mailbox can be seen in Figures 3.2 and 3.3. One word of data can be stored in each mailbox slot. Stored words are accessed in FIFO order. The following mailbox registers are available:

3.6.2 Mailbox Registers

Each Compute Processor has access to a bi-directional mailbox capable of queueing data words. The mailbox state is accessed/modified using the following registers:

TX Mailbox transmit register. Writing to this register will store a data word for retrieval by the corresponding processor.

RX Mailbox read register. Reading this register returns the oldest enqueued word from the mailbox.

TXCNT The number of words that can be transmitted.

RXCNT The number of words waiting to be read.

3.6.3 Intra-processor Barrier

All transfers, even between Local Memory and the VCP register file, are non-blocking. That is, subsequent instructions may be issued before any given transfer has completed. A barrier instruction is provided that stalls the processor pipeline until all pending transfers have completed.

3.7 Data Movement Instructions

This section describes the DPA instructions related to data movement.

`MEM2LM` DP transfers data from Main Memory to Local Memory. Transfers are consecutive groups of vectors.

`LM2MEM` DP transfers data from Local Memory to Main Memory. Transfers are consecutive groups of vectors.

`LM2VPR` CP transfers data from Local Memory to the VCP register file. Multiple consecutive registers may be loaded by this instruction. Local Memory can be accessed at stride.

`VPR2LM` CP transfers data from the VCP register file to Local Memory. Multiple consecutive registers may be read. Local Memory can be written at stride.

`IBARRIER` Blocks the processor pipeline until all pending transfers issued on by the processor have completed.

3.8 Control/State Register Access

This section describes the DPA instructions related to control/state register access.

`CP2DSR, DP2DSR` Sets the following control/state registers.

```
CWCNT, CRCNT, DWCNT, DRCNT, TX
```

`DSR2CP,DSR2DP` Provides read access to the following control/state registers.

```
CWCNT, CRCNT, DWCNT, DRCNT, RX, TXCNT, RXCNT
```

`CP2VSR` Sets VCP control/state registers.

`VSR2CP` Provides read access to VCP control/state registers.

3.9 SIMD Instructions

This section describes the DPA instructions that perform vector operations. These instructions are issued by the CP and executed on the Vector Co-Processor. Here the term packed represents operation over one vector register that is "packed" with multiple scalar elements. Unpack operations are used to shuffle scalar elements among the vector registers.

`ADDP` Packed add.

`SUBP` Packed subtract.

`MULP` Packed multiply.

`UNPCKHP` Interleaved unpack of high-order values.

`UNPCKLP` Interleaved unpack of low-order values.

`UNPCKEP` Unpack of even-order values.

`UNPCKOP` Unpack of odd-order values.

`SPLAT` Broadcasts a single value to all elements in the target.

`REVS` Reverses the vector, maintaining the position of the highest-order element.

`PALIGNR` Packed align right.

3.10 DPA Hardware Implementation

The DPA system has been implemented in the form of a VHDL/Verilog design. This design can be simulated in software using HDL/RTL simulation technologies in ModelSim [10] for testing, verification, and cycle-accurate timing. In addition, the

DPA has been synthesized to an Field Programmable Gate Array (FPGA) system for additional testing and verification [13].

3.11 Programming Considerations

This section discusses the major considerations when programming the DPA [13].

3.11.1 Main Memory Transfer Scheduling

In architectures where the on-chip memory is utilized as a cache, data is loaded as a side-effect of the memory access pattern. With the DPA, the programmer has explicit control over transfers between Main Memory and Local Memory. Therefore, it is the programmer's responsibility to maximize the use of off-chip bandwidth by scheduling Data Processor transfers.

To maximize bandwidth utilization, attention must be paid to the length and frequency of Data Processor transfers. The DPA's off-chip bandwidth utilization will be maximized by issuing fewer transfer instructions with longer transfer lengths. That is, large transfers are able to utilize nearly the entire available off-chip bandwidth so they should be favored over small, frequent transfers.

3.11.2 Local Memory Management

Using the Data Processor, the programmer has complete control over the contents of Local Memory. In addition, transfers between Main Memory and Local Memory are executed independently of all computation on the Compute Processors. Fully optimized software will schedule Data Processor transfers such that the data available in Local Memory is the working set required for the Compute Processors to be able to perform calculation.

The DPA hardware does not provide data coherence at any point in the system,

therefore it is the programmer's job to ensure coherence or to schedule computation such that each data element is only worked on by one computational unit at a time.

Double buffering is one technique in common use to parallelize computation and data transfers. In this technique, one half of Local Memory is used as a buffer for computation, eventually storing the results of the current operation. The other half of Local Memory then is a buffer for pending data transfers, allowing the Data Processor to read previous result and to store the data required for the next stage of computation. Once both the data transfers and the computation are finished, both processors switch buffers.

3.11.3 Synchronization

Synchronization is an important issue on the DPA since all data transfers retire asynchronously. When a transfer instruction is encountered one processor cycle is spent issuing the operation then the processor is free to issue subsequent instructions. Therefore, explicit synchronization is necessary even when loading from Local Memory to the VCP register file. Data independent algorithms will likely be able to use counter and barrier based synchronization across processors, while mailbox communication will likely be necessary when there are data dependencies.

4. DPA Simulator and Assembly Generation

4.1 Simulator Overview

The DPA Simulator is a high-level simulator. The Simulator functions at the source code level in comparison to other processor simulators which typically function at the binary/machine code level. The Simulator is capable of simulating every architectural component described in the DPA ISA [18]. Architectural components of the DPA are represented and simulated at a high level, meaning they are implemented as data structures and functions written in the Simulator's implementation language, C. The DPA simulator is provided to the user in the form of a software library which implements the features of the DPA system as well as the functionality specific to the simulation, such as configuration file parsing and log generation. No effort is made to simulate the SPARC components or logic in the DPA system. This does not affect the functional accuracy of the simulation but is a consideration for the Performance Model (see Chapter 5 on page 34).

The high level design and implementation of the simulator provides several key advantages over low level simulators.

- The Simulator is easy to modify and extend in order to accommodate changes to the hardware design and the addition of new architectural components.
- Execution time of the simulation is greatly reduced (one order of magnitude or more compared to gate level simulators).
- Verbose feedback can be generated in the form of log files and instruction traces for consumption by the user or other software tools.

4.2 Simulator Design Detail

Every architectural component in the DPA is represented in the simulator. This section describes, in detail, how these components are represented.

4.2.1 Overall Software Design

The overall architecture of the Simulator is that of a software library. The implementation language for the Simulator is C, allowing the interface to the simulator to be defined by a series of C header files. These header files expose the functionality of the simulator to the user and define the public types, data structures, and functions.

The DPA Simulator is configurable at two points in time: compilation and runtime. Compile time configurations are related to the data types used within the simulator (for example, the data type of vector elements) as well as some internal features of the simulator. Typically, it is enough to generate a small set of libraries that cover the configurations of interest to the user. Then, the user can select the correct library when compiling the DP/CP code. Runtime configuration allows the DPA system parameters not configured during library compilation to be set.

Users writing code for use with the Simulator and, ultimately, an instantiation of the DPA in hardware must provide two main functions to be linked with the Simulator library. One main function will contain the operations for the Data Processor while the other will be used for the Compute Processor(s). This concept is similar to the standard C convention of having a function named “main” as the entry point for the program.

To create a Simulator executable, the code for the DP and CP is compiled and linked with the Simulator library. The result is a binary for the host platform that, upon execution, will simulate the execution of the code provided by the DP and CP main functions.

4.2.2 Processor Simulation

DPA processors are simulated using operating system level threads. For each processor in the system, a thread is created such that it will enter the main function for that processor upon execution. Aside from some creation and cleanup operations, each thread will perform only the operations necessary to complete the execution of the functionality of the code implemented for that processor.

Each DPA processor thread simulates only the operations specific to the DPA ISA. Included in this portion of the simulation are all registers and execution units available to the programmer through the ISA. All Vector Co-Processor registers and operations that modify their state are simulated directly. As DPA instructions are encountered in the program's flow they are executed by calling a function provided by the Simulator library.

No operations for the underlying SPARC portions of the chip are simulated directly by the simulator. Instead, these operations are carried out by the host system. Operation identical to a true SPARC system for the portion of the code that would compiled to SPARC instructions is only guaranteed in so far as the base compiler's (a gcc based cross compiler [1]) ability to compile the C code implementation of the DP and CP's main functions to have the same meaning on the host system as it would on a SPARC system. Different compilers, versions, and/or optimization settings may elicit this problem. However, in practice, we have not encountered this issue.

The DPA Simulator library provides internal synchronization only to the extent that its data structures are able to remain consistent. No effort is made to synchronize DPA processor threads such that they execute in lock-step or any other order with respect to each other. The only inter-processor synchronization that is respected by the simulator is that which is exposed through the DPA ISA. If the user code executing on the DP or CPs does not synchronize, then the order of execution of one

DPA processor thread with respect to another will be nondeterministic.

4.2.3 Memory Simulation

The DPA's Main Memory and Local Memory state is included in the simulation. Memory size and segmentation descriptions are part of the Simulator's configuration parameters. In addition, configuration allows the user to specify which Compute Processors are able to access Local Memory segments created in the system. Any configuration from full $M : N$ to $1 : 1$ access is supported. The DPA ISA specifies that these memories may not be accessed directly by the DP or CP, meaning that data stored in either memory bank cannot be transfered to the registers of either processor. Data in either memory bank must be transferred to the Vector Co-Processor register file in order to be operated on.

The design of the Simulator's systems allows for more flexibility. The contents of the Main Memory and Local Memory is exposed by the simulator in the form of a pointer to an array. Access in this manner is not correct with respect to the DPA ISA, but it can be useful to the programmer for the development and testing phases of DPA code creation.

4.2.4 Instruction Simulation

The DPA Simulator implements every instruction specified in the DPA ISA as part of the Simulator library. DPA processor threads simulate an instruction issue by calling the function implementing the instruction in the DPA Simulator library. The instruction implementation is then responsible for modifying the state of the simulator to emulate the operation of the instruction that was called. In addition, any log output or other bookkeeping operations are taken care of at this time.

Instructions issued on a single DPA processor are processed serially by the Simula-

tor. The Simulator will not queue operations for later execution. Instead, each DPA instruction's simulator function will carry out the entire operation before returning to the caller. This design decision sacrifices some functional accuracy in order to increase design and implementation simplicity for the Simulator.

DPA Synchronization instructions are simulated by using the implementation's threading library synchronization support. For inter-processor synchronization, mutex objects are used to create atomic regions while condition variables are used to signal processor threads of state changes. The Simulator does not need to implement intra-processor synchronization since every operation on a single processor is executed serially, as discussed above.

4.3 Assembly Generation

The Simulator library package supports assembly generation for all instructions in the DPA ISA plus any additional user functions supplied by the Simulator. For each instruction in the ISA, a definition is created in the Simulator's interface header files. When configured to build a Simulator binary, these definitions are linked to functions in the Simulator library. When configured to produce DPA compatible binaries, these definitions are implemented by inline assembly. The result is a mixed C/assembly code that can be cross compiled for the DPA hardware. In this way, the same DP/CP C code can be compiled once for use with the simulator and compiled a second time to produce a binary targeted for the DPA hardware.

4.4 Binary Encoding

The standard compilation tool chain for the DPA uses GCC configured to cross compile for the SPARC V8 elf platform [1]. This compiler does not recognize the assembly statements for the DPA ISA. Therefore, it was necessary to develop a tool

to encode DPA assembly to binary machine code. The tool that was created accepts input containing assembly statements and produces output where the instructions from the DPA ISA are replaced with binary literals. The assembler simply copies these literals into the encoded output. The result is an augmented assembly file that can then be assembled using the normal assemblers included with the GCC cross compiler.

4.5 Simulation Logs

The Simulator is capable of creating two types of logs during execution: a simulation log and an instruction trace log. The simulation log is intended to be read by the developer. It contains entries for all DPA instructions processed during the simulation. In addition, it provides some feedback of the system state, such as register values, data values for transfers, and memory contents. The trace log contains the instruction trace for the simulation. Instructions are listed with their parameters as well as the information necessary to follow inter-processor synchronization (namely, the values of transfer counters). The trace log is formatted such that it should be easily machine readable, allowing off-line analysis to be performed by tools that are not part of the simulator proper.

4.6 Verification and Testing

The DPA Simulator has been tested for correctness in two ways. The first is through a series of blackbox tests. These tests invoke instructions in the DPA ISA and then probe the state of the system to ensure that the operation executed correctly. There is at least one test for every instruction in the ISA. Many test suites are generated to cover the set of possible simulator configurations.

The second type of test is verification of simulation results. In this case, the

contents of memory at the end of the simulation are considered to be the final result of the simulation. Tests of this type were conducted using code implementing the WHT and FFT. The memory contents at the end of the simulation is compared to the known correct values to determine if the simulation was successful. In addition, this technique was used to test the DPA Simulator's assembly and machine code generation systems. DPA binaries produced by this system were executed using RTL Simulation, which produced the final state in the simulation's DPA Main Memory.

4.7 Simulation Execution Time

One of the key benefits of the DPA Simulator when compared to logic or gate level simulations is decreased execution time. The DPA Simulator can be much more efficient in terms of execution time due to simulation at a high level. In addition, the Simulator is able to utilize multiple processors/cores on the host system to allow parallel execution of simulation operations. This ability is due to the Simulator's use of multiple operating system level threads for the simulation of the DPA's set of processors. Generating the very verbose logs contributes largely to the DPA Simulator's overall execution time. For this reason, the DPA Simulator will not produce logs by default. The user may activate logs using a command-line argument.

For comparison, the execution times required to run simulations in the HDL simulator ModelSim [10], were recorded. Table 4.1 shows the ratio of ModelSim execution times to DPA Simulator execution times. All experiments were run independently on the same machine. Each row in the table lists the simulation that was executed (the transform and its input size), the ModelSim to DPA Simulator ratio with logging turned off followed by the ratio for DPA Simulator execution with logging turned on. The results shown here indicate that the DPA Simulator is three orders of magnitude faster than ModelSim when simulation logs are not produced and two orders of

Table 4.1: Performance Comparison of ModelSim and DPA Simulator

Simulation	**Ratio**	**Ratio Logs On**
WHT 128k	$1.8 \cdot 10^3$	$6.2 \cdot 10^2$
WHT 1m	$1.7 \cdot 10^3$	$6.0 \cdot 10^2$
DFT 64k	$1.4 \cdot 10^3$	$4.7 \cdot 10^2$
DFT 1m	$1.7 \cdot 10^3$	$5.1 \cdot 10^2$

magnitude faster when logs are produced.

5. Performance Model

5.1 Performance Model Overview

The DPA Simulator includes a performance model for the DPA system. This model is included in the DPA Simulator library discussed in Section 4.2.1. The Performance Model executes as a side effect of the simulation proper. The Model maintains its own internal state and only relies on the Simulator to provide it with the sequence of instructions as they are executed. From this information, it calculates the number of processor cycles for each operation. The final result produced by the model is a set of statistics, including the CPU cycles for each processor and the total execution time.

5.2 Performance Model Implementation

This section describes the implementation of the Performance Model, what is included in the model, and how it is modeled.

5.2.1 Design

At a high level, the design of the Performance Model resembles that of an event driven system. Events are created as instructions are processed by the Simulator. Each event represents an instruction of a certain type being issued on a processor. As the Performance Model handles events, it updates its own internal state and statistics.

Figure 5.1 shows the event processing flow for the Performance Model. In this figure, software components are drawn as boxes. Multiple component instances/objects are depicted by layered boxes. Messages and function calls are shown as lines with arrows indicating the receiver and labels describing their purpose/content. Dotted lines

represent return values. The DPA Simulator and relevant components are depicted on the left while the Performance Model and relevant components are on the right. Here, the Simulator is depicted producing an event for an instruction of a specific type. The event contains information specific to the instruction issued and is nearly equivalent to what would be contained in the assembly syntax for that instruction. For example, a Global Memory to Local Memory load instruction event contains the address in Global Memory and the transfer size.

The Performance Model is notified of an event through its interface, which is made accessible to all Simulator components. This interface defines handler functions for each type of instruction. The Simulator notifies the Performance Model of an event by calling one of these functions. Each handler function is responsible for dispatching the instruction to the appropriate Processor object within the Model using a map from Simulator threads to Processor objects.

Processor objects track the state, including the statistics of interest, for a single processor in the DPA system. When an event is received, the Processor dispatches the event to the appropriate Instruction Model object which calculates and returns the number of cycles required to execute that instruction. Instruction Model objects may query the state of the Processor or the system during their calculations. After receiving the count of cycles from the Instruction Model, the Processor object performs the necessary bookkeeping before retiring the event.

5.2.2 Model Parameters

The Performance Model can be configured by setting parameter values which are read at runtime. Some parameters that affect the simulation as a whole, such as memory sizes, vector lengths, and number of compute processors are also read by the Performance Model. Tables 5.1, 5.2, and 5.3 describe the Performance Model specific

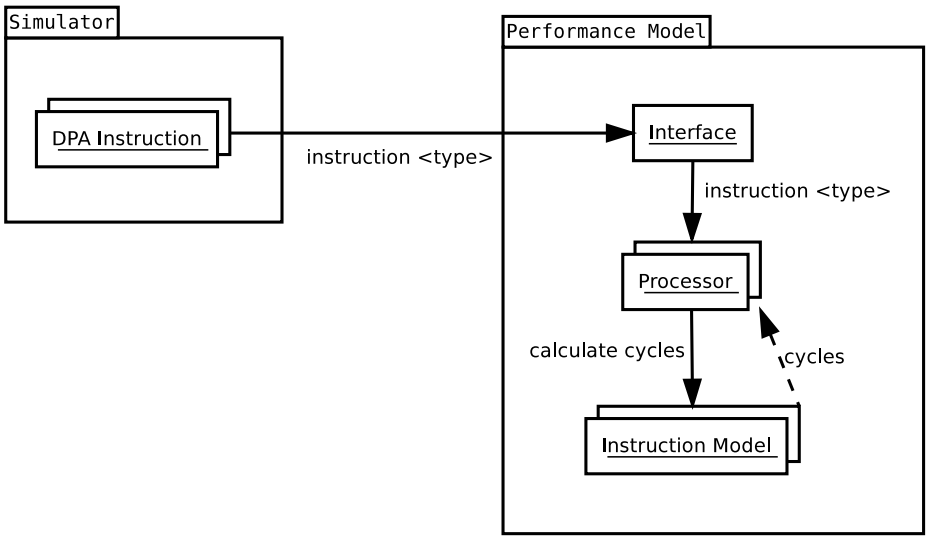

Figure 5.1: The Performance Model instruction event handling.

Table 5.1: Compute Processor Performance Model Parameters

Parameter	**Purpose**
Frequency	Frequency of the Compute Processor in MHz
Vector Transfer Delay	Delay in cycles vector transfer issue and processing
Bytes per Vector	Total vector size in bytes
Wait Overhead	Cycles required to issue a wait operation
Wait Loop Cycles	Cycles for one pass of wait loop
Mailbox Overhead	Cycles required to issue a Mailbox operation

parameters for the CP, DP, and DDR subsystems, respectively.

5.2.3 Instruction Models

Each instruction type has a cost model associated with it, where cost is measured in cycles. This section describes the models used to calculate the number of cycles for individual instructions.

Table 5.2: Data Processor Performance Model Parameters

Parameter	**Purpose**
Frequency	Frequency of the Data Processor in MHz
Wait Overhead	Cycles required to issue a wait operation
Wait Loop Cycles	Cycles for one pass of wait loop
Mailbox Overhead	Cycles required to issue a Mailbox operation

Table 5.3: DDR Performance Model Parameters

Parameter	**Purpose**
Frequency	Frequency of the DDR bus
Bytes Per Cycle	Total bytes delivered each DDR cycle
Bytes Per Vector	Bytes required to transfer one DPA vector
Page Address Length	Number of bits required to describe a page address
Page Switch Cycles	Cycles required to access a new DDR page
Efficiency	Transfer efficiency considering hardware refresh rates
Refresh Threshold	Cycles until a DDR refresh is required

Global Memory Transfers

Memory transfer cycles are calculated in terms of DDR cycles which are then translated into CPU cycles. The model for the number of DDR cycles is:

$$\frac{s \times vs}{w} + pcost(a) \tag{5.1}$$

where s is the transfer size in vectors, vs is the number of bytes per vector, w is the bandwidth in bytes per cycle, and $pcost$ is the page access cost, in cycles, which is a function of the page address, a.

If the number of DDR cycles is greater than the DDR refresh threshold, then the cycle count is scaled by the DDR efficiency:

$$c_e = \frac{c}{e} \tag{5.2}$$

where c is the cycle count calculated by 5.1, e is the DDR efficiency expressed as a percentile, and c_e is the scaled cycle count. DDR efficiency must be considered due to the properties of DDR refresh.

The resulting DDR cycle count is then converted to CPU cycles:

$$c_{cpu} = c_e \times b \tag{5.3}$$

where b is bus multiplier, i.e. the ratio of CPU to DDR cycles.

Local Memory to Vector Register File Transfers

Transfers from Local Memory to the Vector Register File are calculated by:

$$c = s + d \tag{5.4}$$

where s is the number of vectors transferred and d is the vector transfer delay. After this value is calculated and returned to the processor, it is not immediately charged to the current count of CPU cycles. Instead, the cycle at which the transfer will complete is tracked such that it can be referenced by later synchronization instructions, if necessary. This reflects the non-blocking nature of the DPA's transfer instructions in relation to other instructions issued on the Compute Processor.

Intra-Processor Barrier

The `IBARRIER` instruction is used to block the processor's pipeline until all pending transfer instructions complete. This behavior is modeled by tracking the processor cycle at which the most recent transfer will complete, c_t. When a barrier instruction is handled, this cycle is compared against the current cycle, c, for the processor. If the c_t is in the future, the barrier will then consume $c_t - c$ cycles. If c_t is in the past,

the barrier will consume a single cycle.

5.2.4 Vector Operations and Other Instructions

All remaining DPA instructions, aside from inter-processor synchronization instructions, explained below, use a constant execution time of one cycle for their models. This includes all Vector Register File, SIMD, and read/write operations to status registers. This aspect of the model matches the single issue nature of the DPA's processors.

5.2.5 Inter-Processor Synchronization

The DPA supports two methods of Inter-processor Synchronization: blocking until counter update and blocking until a message is received through the FIFO mailbox. Both types of synchronization are modeled in a similar manner.

When any processor updates a counter or writes to a mailbox, the time of the write is recorded in a map of write operation to cycle. The map stores the counter value or mailbox entry id, respectively, and the cycle at which the update occurred. This structure allows the time of the state change to be accessed directly from the counter value or mailbox entry.

When a wait statement or mailbox read is processed, update time is found in the respective map. The time of the update, denoted as c_u is compared to the current cycle time, c. If this time is in the future, that is $c_u > c$, then the synchronization operation is considered to take $(c_u - c) + h$ cycles, where h is the overhead associated with processing either the wait or mailbox read operations. If this time is in the past, then the synchronization operation consumes h cycles.

In the current version of the DPA interface software, counter wait operations are implemented by a loop that polls the counter value. To accurately model this aspect

of the implementation, the value $(c_u - c) + h$ as calculated above, is modified such that it is greater than $(c_u - c)$ cycles and is a multiple of Wait Loop Cycles, a Performance Model parameter described in Table 5.1.

5.3 DP and Memory Transfer Accuracy

This section shows the accuracy of the DPA Simulator's Performance Model for transfers from Global Memory to Local Memory using a set of transfer benchmarks. Here, the estimated times produced by the Performance Model are compared to the cycle accurate measurements taken from ModelSim running an HDL implementation of the DPA system. The Simulator was configured with the same parameters as the HDL used with ModelSim.

The first set of benchmarks compares the estimated performance to the actual performance for transfers between Global Memory and Local Memory of increasing size. This benchmark was constructed by performing a fixed number of iterations of transfers of the same size to increasing addresses. The time per transfer was calculated by $\frac{\text{total transfer time}}{\text{iterations}}$. The results are shown in Figures 5.2 and 5.3 for Global Memory to Local Memory (`MEM2LM`) and Local Memory to Global Memory (`LM2MEM`) transfers, respectively. The $\log_2$ of the transfer size is shown on the X axis and the $\log_2$ of the transfer time in nanoseconds is shown on the Y axis. The average relative difference between the predicted times and the true times is 14.72% for `MEM2LM` and 25.27% for `LM2MEM`.

The deviation between the predicted times and the actual times for `LM2MEM` transfers smaller than 2^4 is due to not completely modeling the hardware's transfer queues. Here, the queue hides some of the transfer cost for very small size `LM2MEM` transfers. This causes the measured transfer times to be lower than the Performance Model's predicted transfer times.

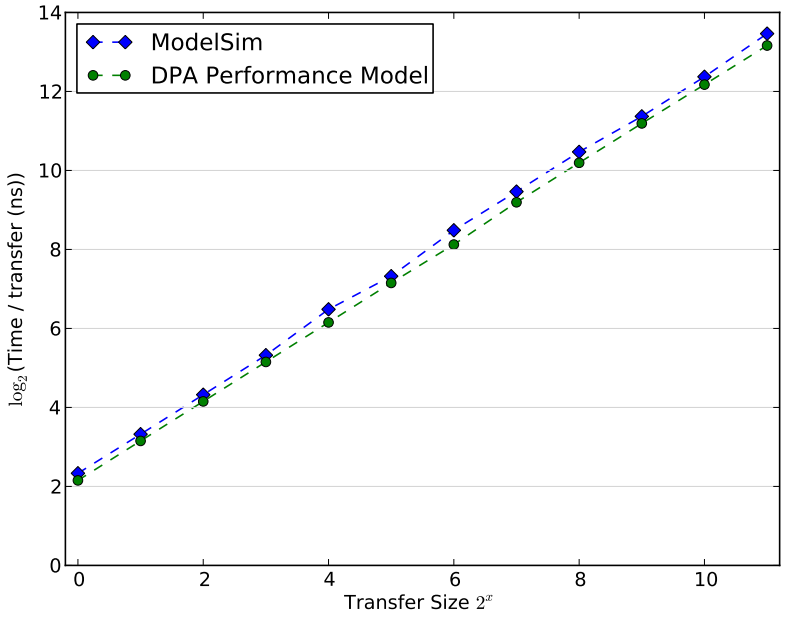

Figure 5.2: MEM2LM Transfer Predicted Performance

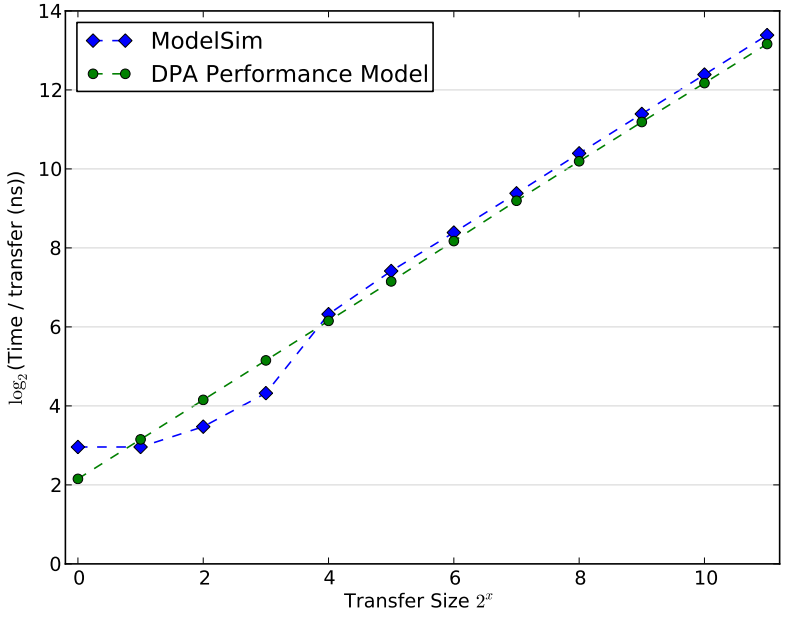

Figure 5.3: LM2MEM Transfer Predicted Performance

The second set of benchmarks compares the estimated performance to the actual performance for transfers with varying numbers of writes per DDR page. In the DDR implementation used in the DPA system, accessing a new page has a fixed cycle cost. This implies that bandwidth will be more efficiently used if transfers are able to access the same page repeatedly. This benchmark was constructed by performing many iterations of transfers where the source Global Memory address is increased by a constant stride after every transfer in the loop. As the stride decreases, the number of writes per DDR page increased.

The results for this benchmark are shown in Figures 5.4 and 5.5 for `MEM2LM` and `LM2MEM`, respectively. The $\log_2$ of the number of consecutive writes per DDR page is shown on the X axis while the $\log_2$ of the time per transfer in nanoseconds is shown on the Y axis. The plots display the time required for one transfer of size 2 and 64 as the writes per page increases along the X axis. As expected, the time per transfer decreases as the writes per page increases. The average relative difference between the predicted times and the true times for `MEM2LM` size 64 is 15.00%, for size 2 is 16.084%, and for `LM2MEM` size 64 is 18.86% and for size 2 is 14.43%.

5.4 CP Accuracy and SPARC Instruction Accounting

Instructions issued on the Compute Processor fall into four general categories: transfers between Local Memory and Vector Register File, Vector operations, synchronization, and scalar operations defined in the SPARC V8 ISA [4]. The Performance Model is capable of handling all categories, except for SPARC instructions.

In DPA code, SPARC instructions are used mainly for Local Memory address calculation and general control flow (loops, function calls, etc.) operations. The impact of these instructions can, in general, be thought of as overhead that reduces the utilization of the vector transfer and computation units. While the Performance Model

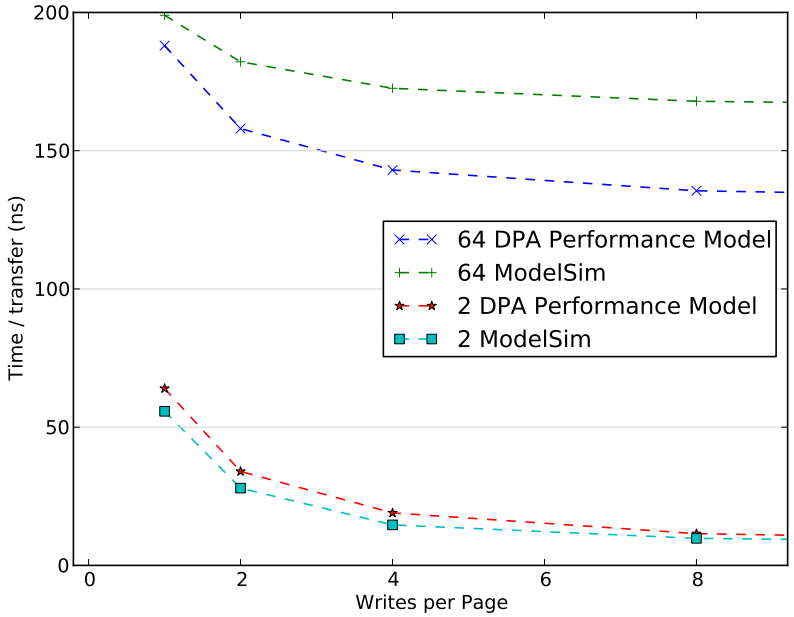

Figure 5.4: MEM2LM Strided Access Predicted Performance

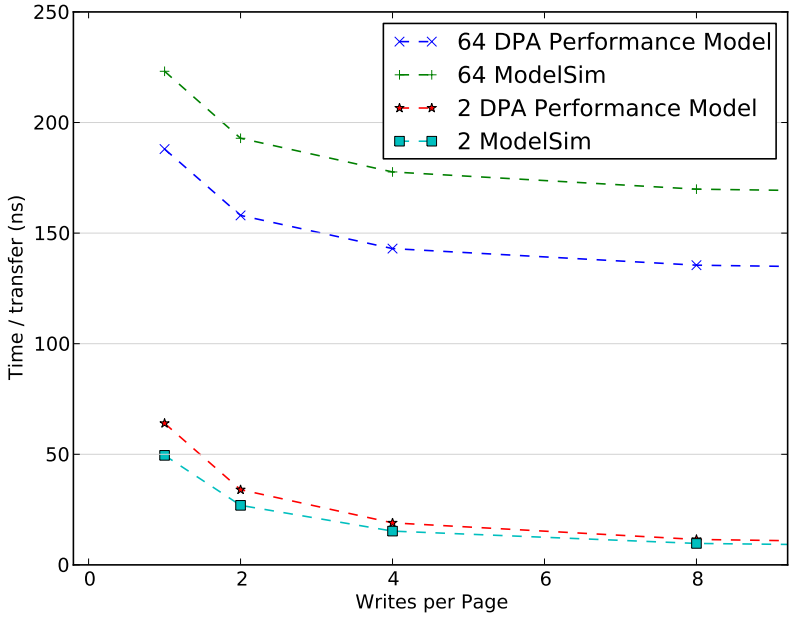

Figure 5.5: LM2MEM Strided Access Predicted Performance

cannot directly estimate the cost of these instructions from the simulation, a method has been implemented that allows the user to inform the model of the SPARC instructions that are executed at any point in the DPA instruction stream. This functionality allows the user to annotate the DPA code with statements indicating the number of SPARC instructions that will be executed before subsequent DPA instructions will be issued. In the current implementation, each SPARC instruction is assigned a cost of one CPU cycle. These annotations can be used to increase the accuracy of the Performance Model's estimate for CP computation time. In code where the execution time of SPARC instructions dominates over the time of DPA instructions, use of these annotations will be necessary for high accuracy. The remainder of this section quantifies the effect of these annotations for a set of benchmark code.

The benchmark used for the following experiments is based off of a simple vector addition where $C[i] \leftarrow A[i] + B[i] \ : \ 0 < i < N - 1$ where N is the vector length. Figure 5.6 shows the DPA code for the Compute Processor used to implement this. For the code shown here, the data is assumed to be already available in Local Memory. Then, the data at each index is loaded into the Vector Register File, operated on, and stored back into Local Memory. Figure 5.7 shows this code annotated with SPARC instruction counts (line 4 and so on). These annotations account for the SPARC instructions associated with address calculations (code of the form `&A[i]`) and overhead due to the for loop. The Performance Model estimated time and the true time, as measured using ModelSim, to execute this benchmark can be found in Table 5.4 where the code in Figures 5.6 and 5.7 is labeled Basic and Annotated Basic, respectively. Here the difference between the estimated execution time and the true time for the Basic version is about -30% while the Annotated version is less than 1%. This shows that the difference in the Basic version is due to not accounting for the SPARC instructions.

The vector addition benchmark was then modified to unroll the for loop by a factor of eight. A portion of the annotated version of the code can be seen in Figure 5.8 and the timing results in Table 5.4 for labels Basic Unrolled and Annotated Basic Unrolled. Unrolling the code has the effect of reducing overhead due to the for loop. This can be seen as a reduction in the execution time in ModelSim from the Basic version to the Basic Unrolled version. The Performance Model estimated execution time for the Unrolled version does not change compared to the original Basic version, since the quantity and sequence of DPA instructions is not modified by unrolling the loop. For the code annotated with SPARC instruction counts, the difference is about -1.6%.

The basic vector addition algorithm was then written using software pipelining, allowing some transfers and vector operations to occur in parallel. The loop body for this code can be seen in Figure 5.9 and the timing results in Table 5.4 for labels Pipelined and Annotated Pipelined. In this case, pipelining the code in this manner increases the execution time due to the use of wait statements, which are more expensive to execute than barrier statements. In addition, more SPARC instructions are needed per vector computation. For this version, the Performance Model's predictive capabilities without accounting for SPARC instructions is poor: the difference between the estimated and true times is -49.35%. However, when this code is annotated with SPARC instruction counts, the difference drops to -14.89%.

The final version of the benchmark code unrolls the pipelined version by a factor of eight. The loop body for this code can be seen in Figure 5.10 and the timing results in Table 5.4 for labels Pipelined Unrolled and Annotated Pipelined Unrolled. In this version of the code, unrolling allows all synchronization statements to be removed from the loop body, resulting in the fastest version of the code. However, each vector operation has a series of address computations associated with it that are

Table 5.4: CP Vector Add Benchmark Results

Code Version	**Simulator Time (us)**	**ModelSim Time (us)**	**Relative Difference**
Basic	1.56	2.29	-31.5%
Annotated Basic	2.27	2.29	-0.646%
Basic Unrolled	1.56	2.04	-23.45%
Annotated Basic Unrolled	2.01	2.04	-1.59%
Pipelined	1.99	3.92	-49.35%
Annotated Pipelined	3.34	3.92	-14.89%
Pipelined Unrolled	0.57	1.52	-61.94%
Annotated Pipelined Unrolled	1.43	1.52	-5.40%

not accounted for in the un-annotated version, resulting in a difference of -61.94% between the estimated and true times. When SPARC instruction annotations are added, this difference drops to -5.40%.

The benchmarks described here and the results shown in Table 5.4 serve as a method to quantify the magnitude of error that can be introduced into CP side calculations due to only accounting for DPA instructions and ignoring SPARC instructions. The relative error will vary from code to code and may be large enough to have a significant effect on the estimate for the overall algorithm execution time. In these cases, the accuracy can be improved by annotating the code indicating the number of SPARC instructions that will be executed before DPA instructions.

5.5 Algorithm Simulation Accuracy

This section shows the accuracy of the DPA Performance Model on simulations of two algorithms across multiple, input sizes. The execution times estimated by the Performance Model are compared to the true times measured through HDL simulation of the DPA using ModelSim.

```
1  for(i = 0; i < t; i++)
2  {
3      // Load vector register file
4      LM2VPR(&A[i], 0, 0, 0);
5      LM2VPR(&B[i], 1, 0, 0);
6
7      // Block until loads finish
8      IBARRIER();
9
10     // Computation on registers
11     ADDP(0, 1, 2);
12
13     // Store to LM
14     VPR2LM(2, &C[i], 0, 0);
15 }
```

Figure 5.6: DPA basic vector addition benchmark code listing

```
1  for(i = 0; i < t; i++)
2  {
3      // Load register file
4      SPARC_INS_COUNT(1);
5      LM2VPR(&A[i], 0, 0, 0);
6      SPARC_INS_COUNT(1);
7      LM2VPR(&B[i], 1, 0, 0);
8
9      // Block until loads finish
10     IBARRIER();
11
12     // Computation on registers
13     ADDP(0, 1, 2);
14
15     // Store to LM
16     SPARC_INS_COUNT(1);
17     VPR2LM(2, &C[i], 0, 0);
18
19     // Account for loop overhead
20     SPARC_INS_COUNT(4);
21 }
```

Figure 5.7: DPA annotated basic vector addition benchmark code listing

```
1  for(i = 0; i < t; i+=8)
2  {
3      // Load register file
4      SPARC_INS_COUNT(1);
5      LM2VPR(&A[i], 0, 0, 0);
6      SPARC_INS_COUNT(1);
7      LM2VPR(&B[i], 1, 0, 0);
8
9      // Block until loads finish
10     IBARRIER();
11
12     // Computation on registers
13     ADDP(0, 1, 2);
14
15     // Store to LM
16     SPARC_INS_COUNT(1);
17     VPR2LM(2, &C[i], 0, 0);
18
19     // Load register file
20     SPARC_INS_COUNT(1);
21     LM2VPR(&A[i+1], 0, 0, 0);
22     SPARC_INS_COUNT(1);
23     LM2VPR(&B[i+1], 1, 0, 0);
24
25     // Block until loads finish
26     IBARRIER();
27
28     // Computation on registers
29     ADDP(0, 1, 2);
30
31     // Store to LM
32     SPARC_INS_COUNT(1);
33     VPR2LM(2, &C[i+1], 0, 0);
34
35     // And so on until we operate on index i+7
36     ...
37 }
```

Figure 5.8: DPA annotated unrolled basic vector addition benchmark code listing

```
1  /* Prologue responsible for
2  initial vector load removed */
3  for(i = 0; i < t-2; i += 2)
4  {
5      // Wait for register group 1 load
6      SPARC_INS_COUNT(1);
7      crc += 2;
8      WAIT_CR_INC(crc);
9
10     // Operate on group 1
11     ADDP(0, 1, 2);
12
13     // Store group 1
14     SPARC_INS_COUNT(1);
15     VPR2LM(2, &C[i],0, 0);
16
17     // Load group 1 for next iteration
18     SPARC_INS_COUNT(2);
19     LM2VPR(&A[i+2], 0, 0, 0);
20     SPARC_INS_COUNT(2);
21     LM2VPR(&B[i+2], 1, 0, 0);
22
23     // Wait for register group 2 load
24     SPARC_INS_COUNT(1);
25     crc += 2;
26     WAIT_CR_INC(crc);
27
28     // Operate on group 2
29     ADDP(3, 4, 5);
30
31     // Store group 2
32     SPARC_INS_COUNT(2);
33     VPR2LM(5, &C[i+1], 0, 0);
34
35     // Load group 2 for next iteration
36     SPARC_INS_COUNT(2);
37     LM2VPR(&A[i+3], 3, 0, 0);
38     SPARC_INS_COUNT(2);
39     LM2VPR(&B[i+3], 4, 0, 0);
40
41     SPARC_INS_COUNT(4);
42 }
43 /* Epilogue responsible for
44 final vector store removed */
```

Figure 5.9: DPA annotated, pipelined vector addition benchmark code listing

```
1  /* Prologue responsible for
2  initial vector load removed */
3  for(i = 0; i < t-8; i += 8)
4  {
5       // Operate on group 1
6       ADDP(0, 1, 2);
7       SPARC_INS_COUNT(1);
8       VPR2LM(2, &C[i],0, 0);
9       SPARC_INS_COUNT(2);
10      LM2VPR(&A[i+8], 0, 0, 0);
11      LM2VPR(&B[i+8], 1, 0, 0);
12
13      // Operate on group 2
14      ADDP(3, 4, 5);
15      SPARC_INS_COUNT(2);
16      VPR2LM(5, &C[i+1], 0, 0);
17      SPARC_INS_COUNT(2);
18      LM2VPR(&A[i+9], 3, 0, 0);
19      SPARC_INS_COUNT(2);
20      LM2VPR(&B[i+9], 4, 0, 0);
21
22      // And so on until we operate on index 1+7
23       ...
24 }
25 /* Epilogue responsible for
26 final vector store removed */
```

Figure 5.10: DPA annotated, unrolled, pipelined vector addition benchmark code listing

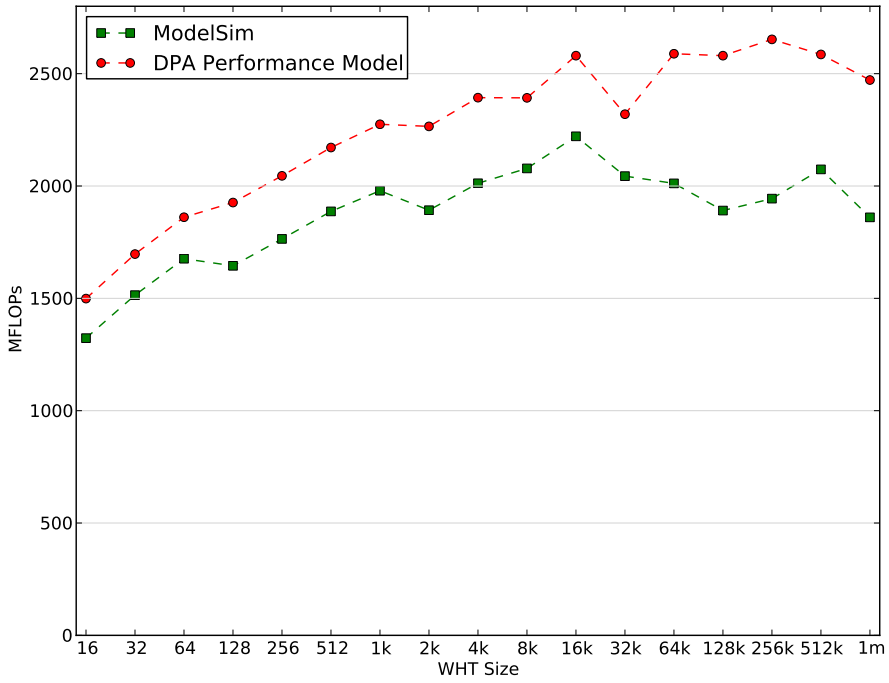

Figure 5.11: WHT Performance estimation compared to ModelSim

5.5.1 WHT Accuracy

Figure 5.11 shows the estimated and true performance of an implementation of the WHT on the DPA. Here, input sizes ranging from 16 to 1 million points are shown on the X axis and the performance measured in millions of floating point operations per second is shown on the Y axis. The average relative difference between the Performance Model and the ModelSim values is 20.1%. The largest relative difference is 36.46% and occurs at input size 128k. For this benchmark, the DPA system was configured with Local Memory sizes ranging from 32 to 128 KB, 1 CP with 64 Vector Registers, and a DP/CP clock of 900 MHz.

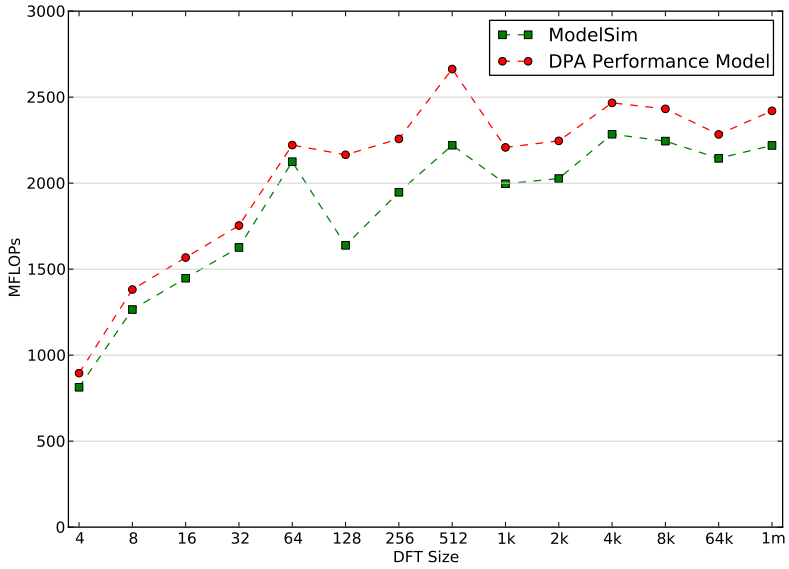

Figure 5.12: DFT Performance estimation compared to ModelSim

5.5.2 DFT Accuracy

Figure 5.12 shows the estimated and true performance of an implementation of the DFT on the DPA. Here, input sizes ranging from 4 to 1 million points are shown on the X axis and the performance measured in millions of floating point operations per second is shown on the Y axis. The average relative difference between the Performance Model and the ModelSim values is 11.5%. The largest relative difference is 32.1% and occurs at input size 128. For this benchmark, the DPA system was configured with Local Memory sizes ranging from 32 KB to 1 MB, 1 CP with 64 Vector Registers, and a DP/CP clock of 900 MHz. The DFT twiddle factors were pre-loaded into Global Memory before the benchmark began.

6. Architecture Exploration

The Data Pump Architecture is intended for use as an application specific embedded processor. In this use case, the DPA's parameters can be selected to match the target application such that the architecture is balanced. A balanced architecture can be considered to be an architecture that provides good performance without excess resource allocation. We can describe a balanced system as one where computation time and off-chip communication time are equal and on-chip memory is the minimal amount such that the former condition still holds. More formally, we can describe the operation of selecting architecture parameters as an optimization problem where we desire to maximize performance while minimizing the resources consumed given the finite set of valid resource configurations. We can observe that the frontier of possible solutions to this optimization problem will provide a set of configurations for a balanced system, such that memory and computational resources are minimized without reducing performance, thus balanced, for each configuration on the frontier. The amount of resources consumed by the architecture can be measured in several ways. One fundamental set of measurements might be power consumption and the physical area required to implement the system in hardware. Alternatively, one can consider the architecture parameters relatively, independent of the hardware implementation, declaring an increase in any parameter to be an increase in resource consumption. Therefore, the resources consumed can be measured directly from the parameters selected; an increase in any parameter from the set of those under consideration will be considered an increase in resource consumption. This optimization problem can then be analyzed by exploring the parameter space and measuring the effect of parameter choice on performance. For the explorations described below, the parameters under consideration are Local Memory size, number of Compute Processors, memory

bandwidth, and processor frequency.

To perform the exploration, we have multiple tools at our disposal. From least granular to most granular, we have a mathematical model based on theoretical limits and idealized algorithms, the DPA Simulator, RTL Simulation, and, finally, execution on hardware. Moving from tool to tool in this order results in increased granularity and accuracy by sacrificing flexibility and execution speed. We see that the DPA Simulator is situated in the middle of this spectrum.

6.1 Exploration with Abstract Mathematical Model

Starting with the least granular yet most flexible tool, the mathematical model, we can analyze a broad set of problems and algorithms. The model used here is a simple model based primarily on the time required for memory transfers and transform calculation [13]. We define Workload(Algorithm) to be the number of operations required by the algorithm and w is the maximum of this function. In addition we define D to be the dataset, and M to be the size of on-chip memory. Transfer time is calculated as

$$T_d = (\frac{D}{B_{peak}} + \text{Page} \times T_{page}) \times 2 = \frac{D}{BW(M/\text{Page})} \times 2 \tag{6.1}$$

where BW is a function of the accesses per DDR page, B_{peak} is the maximum of BW, and T_{page} is the time for accessing a new DDR page. Then the maximum performance for the memory transfers is

$$p_{io} = \frac{w}{T_d}. \tag{6.2}$$

Data computation time is then calculated as

$$T_c = \frac{\text{Workload(Algorithm)}}{N_{fp} \times \text{Freq}} \tag{6.3}$$

where N_{fp} is the number of functional units with Freq as the frequency in the DPA configuration. The maximum performance for computation time is then

$$p_c = \frac{w}{T_c}. \tag{6.4}$$

We define the performance for the entire system as

$$p = \min(p_{io}, p_c). \tag{6.5}$$

In this model, the information provided about the algorithm used is idealized, essentially providing a lower bound on the number of operations required. We can also define the theoretical performance limits for the system irrespective of the algorithm used. For bandwidth we have

$$p_{bandwidth} = \frac{w}{2 \times D/B_{peak}}. \tag{6.6}$$

For computation we have

$$p_{compute} = N_{fp} \times \text{Freq}. \tag{6.7}$$

Total system performance will not exceed the minimum of $p_{bandwidth}$ and $p_{compute}$,

$$p_{system} = \min(p_{bandwidth}, p_{compute}). \tag{6.8}$$

Figure 6.1 shows the results of an analysis using this model for WHT algorithms of increasing input size. The system depicted here was configured with 4 CPs, vector width 4, CPU frequency 1000 MHz, Local Memory size 2^{15} kilobytes and off-chip memory bandwidth of 6,400 MB/s. The dashed yellow line shows the peak perfor-

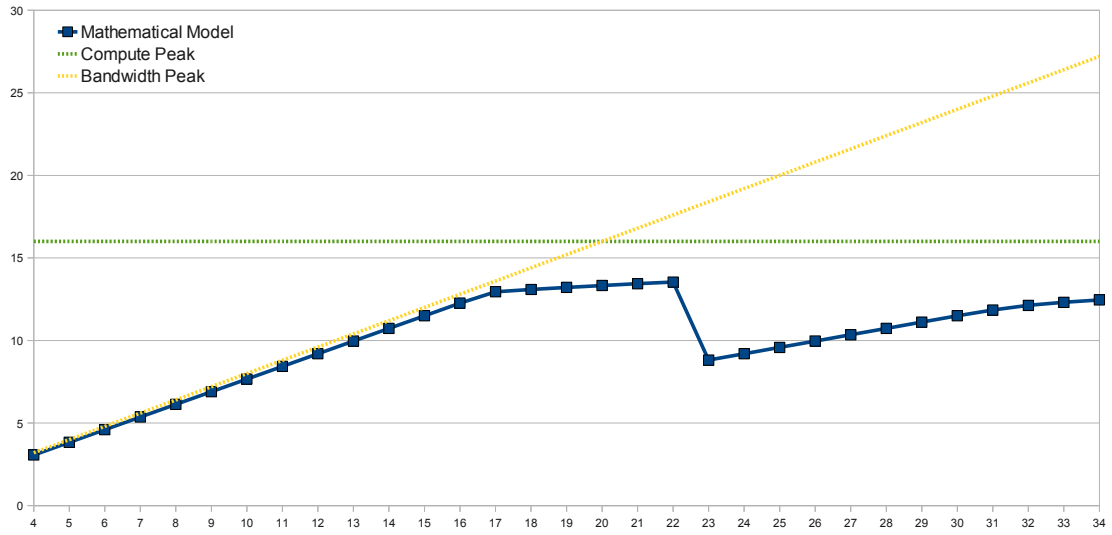

Figure 6.1: WHT Performance for increasing input size from mathematical model

mance when constrained by memory bandwidth while the dashed green line shows the peak performance for when constrained by compute resources. The blue line shows the performance predicted by the model.

One important result of this analysis is the depiction of the crossover point where the performance of the system moves from being memory bound to compute bound. In the plot, this point is where the yellow line crosses the green line and is roughly at input size 2^{20}. We can see that problems smaller than 2^{17} will have their performance limited by memory bandwidth, as indicated by the performance of this points closely following the yellow line. At sizes 2^{17} to 2^{22} the performance is compute bound. Once the problem grows larger than 2^{22}, it becomes memory bound again. This second case of memory bound performance for larger problems is due to the problem size exceeding the capacity of Local Memory and requiring multiple transfers per data element. Then, as the problem size continues to increase beyond this point, the performance slowly trends toward the maximum compute performance. For the

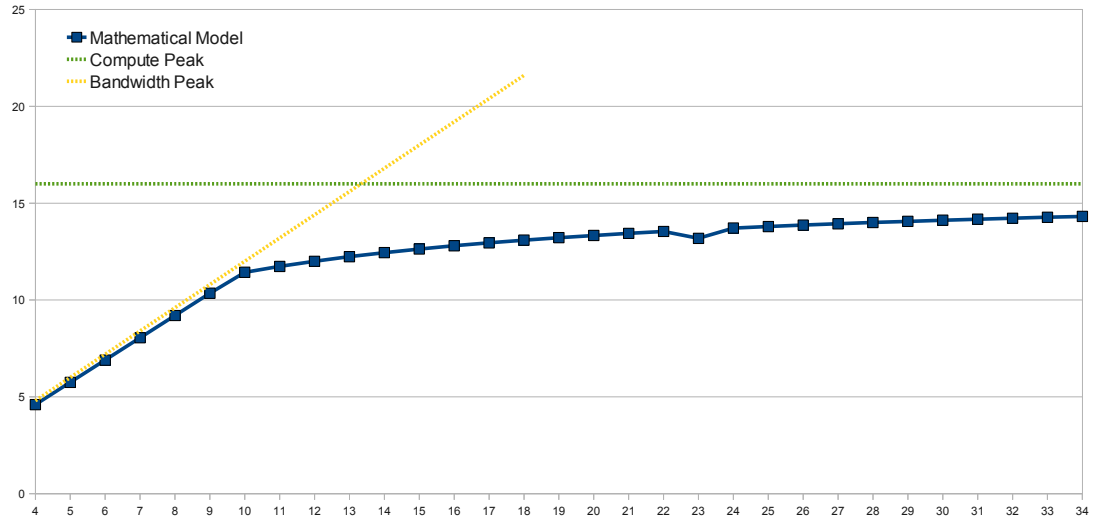

Figure 6.2: WHT Performance for increasing input size from mathematical model configuration with increased memory bandwidth

system depicted in Figure 6.1 we could increase algorithm performance for many input sizes by increasing the available memory bandwidth, which we have done showing the results in Figure 6.2. Here, the memory bandwidth has been increased to 9,600 MB/s. All other parameters of the system remain unchanged. As we see, the cross-over point for this configuration now occurs at roughly 2^{13} and for inputs larger than 2^{10} the performance is compute bound.

The abstract mathematical model shown here allows the designer to consider the problem space at a very high level by providing insight to the trade-off between compute resources and memory bandwidth. Analysis at this level can lead to a first approximation of total system performance for the target application. The designer can adjust configurations to change the point where the performance bottleneck shifts from memory bound to compute bound.

6.2 Exploration with the DPA Simulator

The DPA Simulator is capable of providing greater detail than the idealized mathematical model. However, to be able to use the Simulator, we must first generate code. The initial experiment compares the results using hand-written code to performance estimates using the mathematical model, the DPA Simulator, and ModelSim, shown in Figure 6.3. These results are for a system with 8 CPs, DDR bandwidth of 25,600 MB/s, frequency 3.2 GHz, and 512 KB of Local Memory. Here we see that all three tools follow the same pattern for performance, with the ModelSim curve (shown in green) giving the true performance of this implementation of the WHT for increasing input size. The DPA Simulator consistently over-estimates the performance compared to the true results, however the magnitude of the error is small. In this case, SPARC instruction execution time is not included in the DPA Simulator results which is the most likely cause of the bias. Here, we were able to create an accurate model and hand-write code with performance that matches this model. In the general case, this may not be possible, thus one would expect a larger gap between modeled performance and actual performance as measured using the Simulator or ModelSim.

All of the code for the experiments discussed in the remainder of this section was generated using the SPIRAL system. In order to show more detail, here we have selected a single input size, the WHT of input size $2^{16} = 64k$ as our target application. For each instance of code generated, the SPIRAL system was provided with the selected transform and input size as well as the DPA configuration for Local Memory size and number of Compute Processors. Given this input, the SPIRAL system produced code optimized for these parameters. Modifying the number of Compute Processors and size of Local Memory forces the need for the code to be regenerated in order to utilize the additional resources.

We will start by exploring the trade-off between Local Memory size and the num-

Figure 6.3: WHT Performance for increasing input size showing Model, DPA Simulator, and ModelSim estimates

ber of Compute Processors. As the number of processors increases, each CP will have less Local Memory allocated (either in software or through Local Memory segments) for its use. Increasing either Local Memory or the number of Compute Processors without a mutual increase in the other resource will lead to an imbalance in the system, thus creating bottlenecks that limit the use of the expanded resource. If there are too many Compute Processors and not enough Local Memory, each CP will need to stall as it waits for transfers from off-chip memory into Local Memory. Spilling to off-chip memory will be expensive and can lead to decreased performance.

The results of considering a system for executing the WHT 64k are shown in Figure 6.4. Local Memory size (measured in kilobytes) is in the set $\{2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}\}$, Number of CPs is in the set $\{2, 4, 8, 16\}$, and memory bandwidth and CPU frequency are fixed at 6400 MB/s and 1000 MHz, respectively. Code for each pair in the cross product of these sets was generated. For each pair, the plot displays the performance as a color shade ranging from blue, low performance, to red, high performance. The

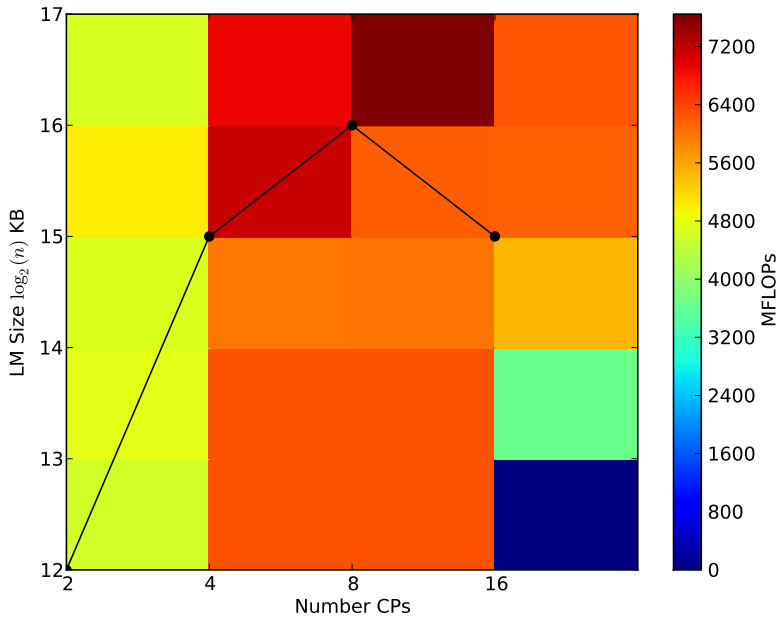

Figure 6.4: Performance evaluation as Local Memory size and Number of Compute Processors varies

color for the level of performance from each pair is expanded to fill the block up to the next parameter value. For instance, the block from 8 to 16 on the x-axis and 16 to 17 on the y-axis is colored for the performance from the point (8,16) corresponding to 8 CPs and 2^{16} kilobytes of Local Memory. The black line and associated points indicates the Pareto frontier. Here, points are included in the frontier such that for each valid choice of number of CPs, the smallest Local Memory size with the highest level of performance is selected. One performance value is considered to be greater than another if it is at least 10% greater. This effectively identifies the boundary after which performance peaks then levels off.

This style of pseudo-color plot has a very intuitive interpretation with respect to the selection of balanced architecture points. All potential configuration points can be found by identifying the lower-left corners of like colored blocks. The coordinates of

this corner identify the configuration parameters. While we have identified the Pareto frontier over the entire range of parameters in Figure 6.4, one can visually identify new frontier points for a subset of the parameters by, for each number of CPs on the x-axis, finding the box with the color signifying the highest level of performance and then finding the lower-left corner of this box.

From Figure 6.4 we can form the conclusion that number of CPs and Local Memory size affect performance jointly and must be considered simultaneously when selecting a configuration of DPA parameters. For instance, we see that if constrained to Local Memory size between 2^{12} and 2^{13} increasing the number of CPs from 4 to 8 does not increase performance while increasing to 16 CPs *decreases* performance. Similarly, when fixing the number of Compute Processors to 4 and looking at the range of LM sizes, we see that increasing LM size beyond 2^{15} provides no increase in performance. In fact, this point has been identified as being on the Pareto frontier. We see that the best performance in this figure occurs when we choose 8 CPs and a Local Memory size of 2^{16}. Figure 6.5 compares utilization rate to the ratio between Local Memory size and number of Compute Processors. Here, utilization rate is defined as the ratio of achieved performance to maximum performance where maximum performance is defined by Equation 6.8. The utilization ratio begins to peak at a ratio of roughly 4,000. Local Memory to number of CP ratios higher than this value show no increase in utilization rate, while ratios less than this exhibit decreased utilization, indicating that balanced systems should target ratios near 4,000 when selecting values for Local Memory size and number of Compute Processors.

Now, we will concentrate on the second pair of parameters, memory bandwidth and compute processor frequency. We can perform exploration of these parameters without additional code generation and only adjusting the configuration provided to the DPA Simulator. The result of this exploration is shown in Figure 6.6 for Local

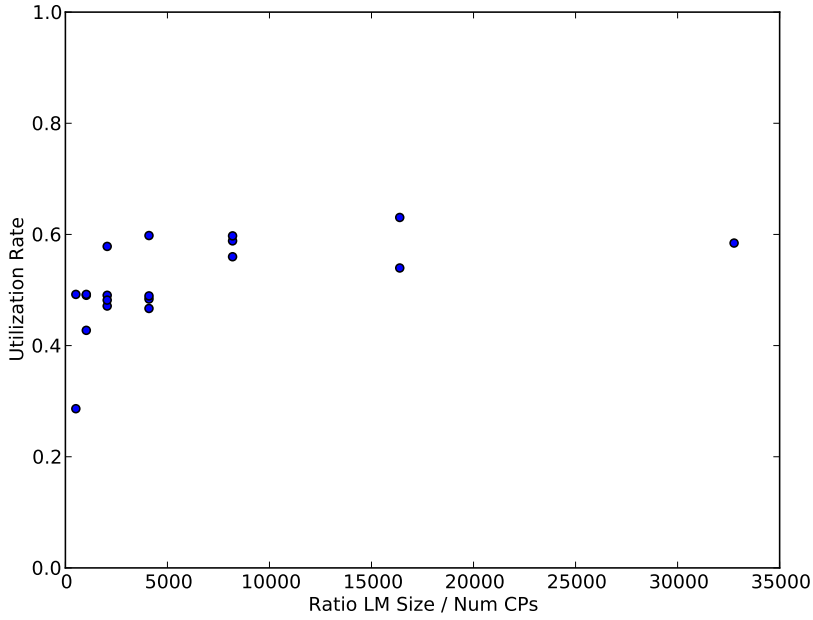

Figure 6.5: Performance for varying ratio of Local Memory size to Number of CPs

Memory size fixed to 2^{15} kilobytes and Number of CPs fixed to 4, a point indicated on the Pareto frontier in Figure 6.4. In this figure, CP Frequency ranges from 600 to 1900 MHz and DDR Bandwidth ranges from 5,000 to 19,000 MB/s. This plot is similar in style to 6.4, here, again, performance is indicated by color with low performance displayed as blue and high performance displayed as red. Also, we indicate the Pareto frontier with the black curve and points along it.

Again, we have the same intuitive interpretation by finding lower-left corners of boxes of the same color to find the minimum resources required to gain that level of performance. As an example, we see from the plot that at a CP frequency of 1100 MHz the minimum memory bandwidth with the best level of performance is at about 13,000 MB/s. We can also note that moving to 1200 MHz increases performance without requiring an increase in DDR bandwidth. These points are shown on the

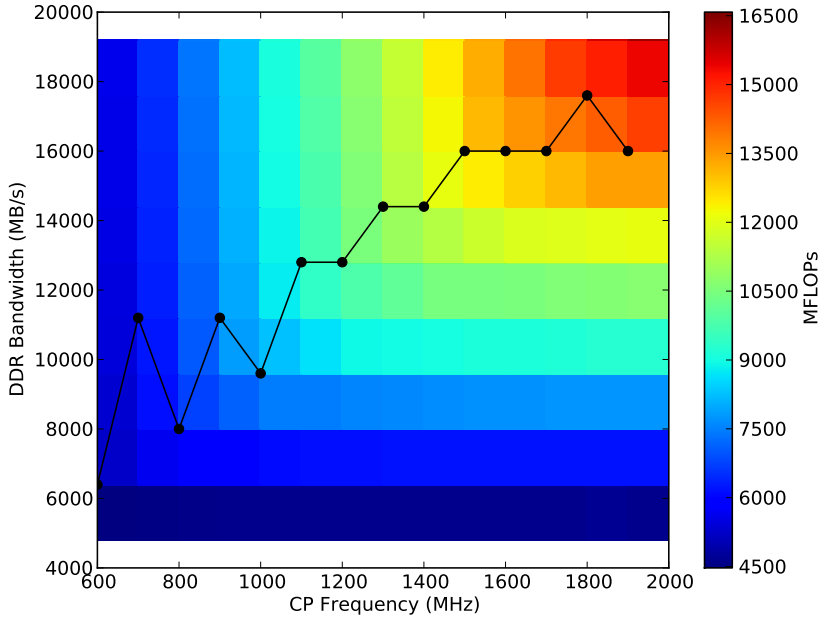

Figure 6.6: Performance evaluation as off-chip Memory Bandwidth and CP Frequency varies

Pareto frontier. The best performance in this Figure occurs when select a frequency of 1900 MHz and a bandwidth of 16,000 MB/s. Figure 6.7 shows the utilization rate compared to ratio between memory bandwidth and CP frequency. Here we see that utilization peaks when the ratio is between 5 and 6. Again, utilization does not increase with ratios larger than this and utilization decreases for smaller ratios. The indication is that bandwidth to frequency ratios between 5 and 6 should be targeted by the designer.

The Simulator allows the designer to perform such experiments and receive performance feedback from a model that is much more similar to that of the actual system compared to the abstract mathematical model used earlier. Exploration at this level forces the designer to use the algorithm and code that would be executed on hardware, ensuring more detail and realism than the idealized algorithms assumed

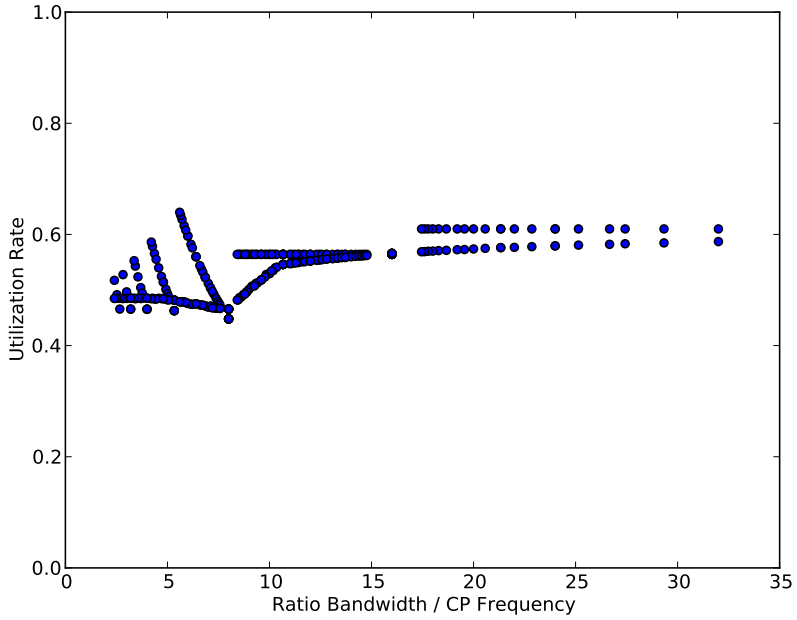

Figure 6.7: Performance for varying ratio of off-chip Memory Bandwidth to CP Frequency

by the mathematical model. After reviewing feedback provided by the Simulator, the designer can then further narrow the set of potential configurations before moving to more granular and time consuming methods of simulation/exploration.

7. Conclusion

The Data Pump Architecture (DPA) is a parameterized, novel non-von Neumann architecture designed for application specific embedded processing. The DPA Simulator and Performance Model is tool that simulates the DPA at a high level, providing fast simulation times, performance feedback, and acceptable accuracy. The DPA Simulator can be used by system designers to explore the DPA parameter space in search of a balanced system for the target application. Such an exploration process, in combination with optimized code generation allows the system, both hardware and software, to be optimized as a whole. Without a tool similar to the Simulator and Performance Model described here, this process would be forced to use abstract and less detailed results from a mathematical model or time consuming and inflexible RTL simulation.

Future work for the DPA Simulator and Performance model includes increasing accuracy through finer tuning of the performance model and the choice of model parameters. Potential work includes the development of an automatic method of parameter adjustment and optimization. In addition, a more accurate method of accounting for SPARC instructions is desirable. The current system relies on code annotation to provide the DPA Simulator with information about SPARC instructions. In this thesis, these annotations were be provided by hand. An improved approach might include SPARC annotations as part of the SPIRAL code generation system or use some new tool to generate annotations automatically. To fully explore the space of DPA configurations, improved algorithm and code generation tools will be needed. Such tools must be capable of analyzing and searching the DPA configuration space in a complete manner. Given these tools, the DPA Simulator can be used as a platform for further architecture exploration by adding new implementations of

DPA functional units with modified or new capabilities. Then, a designer may choose from these functional unit implementations to create a system with capabilities beyond those found in the current DPA ISA and, perhaps, leading to new research in architectural components for use within the DPA system.

Bibliography

[1] Aeroflex Gaisler AB. Leon Bare-C Cross Compilation System. http://www.gaisler.com/cms/index.php?option=com_content\&task=view\&id=147\&Itemid=31.

[2] Aeroflex Gaisler AB. Leon3 Processor. http://www.gaisler.com/cms/index.php?option=com_content\&task=view\&id=13\&Itemid=53.

[3] J. Emer, P. Ahuja, E. Borch, A. Klauser, Chi-Keung Luk, S. Manne, S.S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: a performance model framework. *Computer*, 35(2):68 –76, feb 2002.

[4] SPARC International Inc. *The SPARC Architecture Manual Version 8*. SPARC International Inc., 1992.

[5] Intel. Intel Integrated Performance Primitives (Intel IPP). http://software.intel.com/en-us/intel-ipp/.

[6] J. Johnson, R.W. Johnson, D. Rodriguez, R. Tolimieri. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. In *Circuits, Systems, and Signal Processing 9*, pages 449–500, 1990.

[7] Jeremy Johnson and Markus Püschel. In Search of the Optimal Walsh-Hadamard Transform. In *Proc. ICASSP*, pages 3347–3350, 2000.

[8] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code Generation for DSP Transforms. In *Proc. of the IEEE special issue on "Program Generation, Optimization, and Adaptation"*, volume 93, pages 232–275, 2005.

[9] Matt T. Yourst. PTLsim x86-64 Cycle Accurate Processor Simulation Design Infrastructure. http://www.ptlsim.org/.

[10] Mentor Graphics. ModelSim. http://model.com.

[11] Parallel Systems Architecture Laboratory. SimFlex. École Polytechnique Fédérale De Lausanne. http://parsa.epfl.ch/simflex/.

[12] Neungsoo Park and V.K. Prasanna. Dynamic data layouts for cache-conscious implementation of a class of signal transforms. *Signal Processing, IEEE Transactions on*, 52(7):2120 – 2134, july 2004.

[13] Qian Yu. Personal communication and meeting notes. Carnegie Mellon University, 2010.

[14] SPIRAL. `http://www.spiral.net`.

[15] SPIRAL. DPA ISA V0.2 Addendum 1 Redefine CP Side Data Transfer Instructions.

[16] SPIRAL. DPA ISA V0.2 Addendum 2 Blocking Instructions for Internal Barrier.

[17] SPIRAL. DPA ISA V0.2 Addendum 3 DSR Access.

[18] SPIRAL. DPA Instruction Set Architecture V0.2 Basic Configuration. 2008.

[19] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. How to Write Fast Numerical Code: A Small Introduction. In *Proc. Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4959 of *Lecture Notes in Computer Science*, pages 116–131. Springer, 2008.

[20] Wind River. Virtutech Simics. `http://www.virtutech.com/products/`.

[21] Li Zhao, R. Iyer, J. Moses, R. lllikkal, S. Makineni, and D. Newell. Exploring large-scale cmp architectures using manysim. *Micro, IEEE*, 27(4):21 –33, july-aug. 2007.